Part One: Read Me

Design:

My update function first creates a copy of the board

I do not change anything on the board copy.

I go to each cell on the board copy and call a helper function “getCount”

getCount takes a pointer to the board copy, the indexes of the cell in question, and the max height and max width of the board.

getCount returns how many of its neighbors are alive. (This number actually includes itself also but I account for this by subtracting the value of the cell from the return value)

getCount also uses a second helper function which takes the same parameters as getCount but returns the value of the cell or zero if the indexes are out of bounds.

With the result of my getCount I check the conditions stated in the assignment to decide if it should be a 1 or 0.

I then update the original board accordingly. By reading from the copy board and only altering the original board I avoid changing the alive neighbors count of its neighbors when I get to them.

I then free my array copy.

I did not have many major design challenges. I had one implementation challenge where the loop instruction was not working for some reason because it said it could not be stored in a byte. I had to use another jump and compare instruction instead. It does the same exact thing as the loop though. And it is just in one spot.

Space Efficiency:
h = height of board
w = width of board
The space required for the original board is - 32 Bytes (h*w)
I only make a single copy of the board so the total space used is 2 * (32 Bytes (h*w))
The Big O = **O(h*w)**

Time Efficiency:
h = height of board
w = width of board

I loop through the entire board 2 times. 2(n*m). Once to copy it and once to check if cell should be updated/ update it. The operations performed in each loop are all O(1) so the big O time efficiency is **O(h*w)**